An Analysis of the *Shaft* Distributed Denial of Service Tool

39

By Sven Dietrich, Neil Long and David Dittrich

Introduction

This is an analysis of the *Shaft* distributed denial of service(DDoS) tool. Denial of service is a technique to deny access to a resource by overloading it, such as packet flooding in the network context. Denial of service tools have existed for a while, whereas distributed variants are relatively recent. The distributed nature adds the "many to one" relationship. Throughout this analysis, most actual host names have been modified or removed.

Historical overview

Shaft belongs in the family of tools discussed earlier, such as Trinoo, TFN, Stacheldraht, and TFN2K. As in these related tools, there are handler (or master) and agent programs. The general concepts related to these tools can be found in a *Distributed Intruder Tools Workshop Report* held in November 1999 at the Computer Emergency Response Team Coordination Center (CERT/CC) in Pittsburgh, Pennsylvania, USA: http://www.cert.org/reports/dsit_workshop.pdf.

The chronological order of development is: Trinoo, TFN, Stacheldraht, Shaft, and TFN2K. Trinoo, TFN, and Stacheldraht were analyzed in [5], [6], and [7] respectively. TFN2K was recently analyzed in [1].

In the first two months of 2000, DDoS attacks against major Internet sites (such as CNN, ZDNet, Amazon etc.) have brought these tools further into the limelight. A few papers covering DDoS can be found at:

http://packetstorm.securify.com/distributed/
http://staff.washington.edu/dittrich/misc/ddos/
http://www.cert.org/advisories/CA-99-17-denial-of-service-tools.html

Analysis

Shaftnode was recovered, initially in binary form, in late November 1999, then in source code form for the agent. Distinctive features are the ability to switch handler servers and handler ports on the fly, making detection by intrusion detection tools difficult from that perspective, a "ticket" mechanism to link transactions, and the particular interest in packet statistics.

The network:
client(s)→handler(s)→agent(s)→victim(s)
The Shaft network is made up of one or more handler programs (*shaftmaster*) and a large set of agents (*shaftnode*). The attacker uses a telnet program (*client*) to connect to and communicate with the handlers.

A Shaft network is depicted in Figure 1.

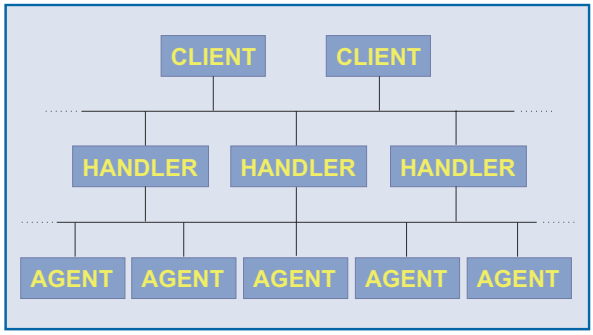

Figure 1 - Shaft Network

Network Communication

Client to handler(s):	20432/tcp
Handler to agent(s):	18753/udp
Agent to handler(s):	20433/udp

Shaft (in the analyzed version, 1.72) is modeled after Trinoo, in that communication between handlers and agents is achieved using the unreliable IP protocol UDP. (See Stevens [18] for an extensive discussion of the TCP and UDP protocols). Remote control is via a simple telnet connection to the handler. Shaft uses *tickets* for keeping track of its individual agents. Both passwords and ticket numbers must match for the agent to execute the request. A simple letter-shifting (Caesar cipher, see Schneier [17]) is also used.

Commands
The command structure is divided into the agent and handler command syntax groups. The attacker interacts with the handler via a command line.

Agent Command Syntax
Accepted by agent and replies generated back to the handler:

`size <size>`
Size of the flood packets.

Generates a "size" reply.

`type <0|1|2|3>`
Type of DoS to run:
0 UDP,
1 TCP,
2 UDP/TCP/ICMP,
3 ICMP

Generates a "type" reply.

`time <length>`
Length of DoS in seconds

Generates a "time" reply.

`own <victim>`
Add victim to list of hosts to perform denial of service on

Generates a "owning" reply.

`end <victim>`
Removes victim from list of hosts (see "own" above)

Generates a "done" reply.

`stat`
Requests packet statistics from agent

Generates a "pktstat" reply.

`alive`
Are you alive?

Generates a "alive blah" reply.

`switch <handler> <port>`
Switch the agent to a new handler and handler port

Generates a "switching" reply.

`pktres <host>`
Request packet results for that host at the end of the flood

Generates a "pktres" reply.

Sent by agent:

`new <password>`
Reporting for duty

`pktres <password><sock><ticket> <packets sent>`
Packets sent to the host identified by <ticket> number

Handler (shaftmaster) Command Syntax
Little is known about the handler, but this is a speculation, pieced together from clues, of how its command structure could look like:

`mdos <host list>`
Start a distributed denial of service attack (mdos = massive denial of service?) directed at <host list>.

Sends out "own host" messages to all agents.

`edos <host list>`
End the above attack on <host list>.

Sends out "end host" messages to all agents.

`time <length>`
Set the duration of the attack.

Sends out "time <length>" to all agents.

`size <packetsize>`
Set the packetsize for the attack (8K maximum as seen in source).

Sends out "size packetsize" to all agents.

`type <UDP|TCP|ICMP|BOTH>`
Set the type of attack:
UDP packet flooding,
TCP SYN packet flooding,
ICMP packet flooding, or
all three (here BOTH = ICMP and IP protocols)

Sends "type <type>" to all agents.

`+node <host list>`
Add new agents

`-node <host list>`
Remove agents from pool

`ns <host list>`
Perform a DNS lookup on <host list>

`lnod`
List all agents

`ltic`
List all tickets (transactions?)

`pkstat`
Show total packet statistics for agents

Sends out "stat" request to all agents.

`alive`
Send an "alive" to all agents.

A possible argument to alive is "hi"

`stat`
show status?

`switch`
become the handler for agents

Send "switch" to all agents.

`ver`
show version

`exit`

Password protection
After connecting to the handler using the telnet client, the attacker is prompted with "login:". Too little is known about the handler or its encryption method for logging in. A cleartext con-

nection to the handler port is obviously a weakness.

Detection

Binaries and their behavior
As with previous DDoS tools, the methods used to install the handler/agent are the same as installing any program on a compromised Unix system, with all the standard options for concealing the programs and files (use of hidden directories, "root kits", kernel modules, etc.) See Dittrich's Trinoo analysis [5] for a description of possible installation methods for this type of tool.

Precautions have been taken to hide the default handler in the binary code. In the analyzed code, the default handler could be defined as follows:

```
#define MASTER "2:3/279/1/2"
```

which would translate into 192.168.0.1 (fictitious address used here) using the same simple cipher mentioned above. Port numbers are munged before actual use, e.g.

```
#define MASTER_PORT 20483
```

is really port 20433.

All these techniques intend to hide the critical information from prying eyes performing forensics on the code. The program itself tries to hide itself as a legitimate Unix process (httpd in the default configuration).

Looking at strings in the shaftnode application reveals the following:

```
>strings -n 3 shaftnode
pktres
switch
alive
stat
end
own
time
type
size
httpd
23:/33/75/28
Unable to fork. (do it manually)
shift
new %s
size %s %s %s %s
type %s %s %s %s
time %s %s %s %s
owning %s %s %s %s
switched %s %s %s
done %s %s %s %s
pktstat %s %s %s %lu
alive %s %s %s blah
%d.%d.%d.%d
Error sending tcp packet from %s:%i to
%lu:%i
pktres %s %i %i %lu
```

Upon launch, the Shaft agent (the *shaftnode*) reports back to its default handler (its *shaftmaster*) by sending a "new <upshifted password>" command. The default password of "shift" found in the analyzed code would thus be "tijgu". Therefore a new agent sends "new tijgu" and all subsequent messages transmit that password. Only in one case does the agent shift in the opposite direction for one particular command, e.g. "pktres rghes". It is unclear at the moment whether this is intentional or not.

Incoming commands arrive in the format:

"command <upshifted password>
<command arg><socket><ticket>
<optional args>"

For most commands, the password and socket/ticket need to have the right magic to generate a reply and the command to be executed.

Message flow diagram between handler **H** and agent **A**:

Initial phase: **A→H**: "new", f(password)
Running loop: **H→A**: cmd, f(password), [args], Na, Nb
A→H: cmdrep, f(password), Na, Nb, [args]

- f(X) is the Caesar cipher function on X
- Na, Nb are numbers (tickets, socket numbers)
- cmd, cmdrep are commands and command acknowledgments
- args are command arguments.

The flooding occurs in bursts of 100 packets per host, with the source port and source address randomized. This number is hard-coded, but need not be. Whereas the source port spoofing works only if the agent is running as a root privilege process, the author has added provisions for packet flooding using the UDP protocol and with the correct source address in the case the process is running as a simple user process. It is noteworthy that the random function is not properly seeded, which may lead to predictable source port sequences and source host IP sequences.

Source port = (rand() % (65535-1024)+1024), where % is the mathematical 'mod' operator. This will generate source ports greater than 1024 at all times.

Source IP = rand()%255.rand()%255.rand()%255.rand()%255

The source IP numbers can (and will) contain a zero in the leading octet.

Additionally, the sequence number for all TCP packets, namely 0x28374839, is fixed. This helps with respect to detection at the network level. The ACK and URGENT flags are randomly set, except on some platforms. Destination ports for TCP and UDP packet floods are randomized.

The client must choose the duration ("time"), size of packets, and type of packet flooding directed at the victim hosts. Each set of hosts has its own duration, which gets divided evenly across all hosts. This is unlike TFN [2], which forks an individual process for each victim host.

For the type, the client can select UDP, TCP SYN, or ICMP packet flooding, or the combination of all three. Even though there is the potential of having a different type and packet size for each set of victim hosts, this feature is not exploited in this version.

The author of Shaft seems to have a particular interest in statistics, namely packet generation rates of its individual agents. The statistics on packet generation rates are possibly used to determine the "yield" of the DDoS network as a whole. This would allow the attacker to stop adding hosts to the attack network when it reached the necessary size to overwhelm the victim network, and also to know when it is necessary to add more agents to compensate for loss of agents due to attrition during an attack (as the agent systems are identified and taken off-line.)

Currently, the ability to switch host IP and port for the handler exists, but the listening port for the agent remains the same. This may, however, change in the future.

A sample attack

In this section we will look at a practical example of a Shaft attack, as seen from the attacking network's perspective. Figure 2 shows what the Unix "lsof" command [10] will show about a shaftnode agent:

```
   # lsof -c shaftnode
COMMAND     PID USER   FD TYPE     DEVICE    SIZE   NODE   NAME
shaftnode 13489 root cwd VDIR      0,0       400       2   /tmp
shaftnode 13489 root txt VREG      0,0     19492      10   /tmp (swap)
shaftnode 13489 root txt VREG     32,0    662764  182321   /usr/lib/libc.so.1
shaftnode 13489 root txt VREG     32,0     17480  210757
/usr/platform/sun4u/lib/libc_psr.so.1
shaftnode 13489 root txt VREG     32,0    566700  182335   /usr/lib/libnsl.so.1
shaftnode 13489 root txt VREG     32,0     39932  182348   /usr/lib/libw.so.1
shaftnode 13489 root txt VREG     32,0     15720  182334   /usr/lib/libmp.so.1
shaftnode 13489 root txt VREG     32,0     15720  182327   /usr/lib/libintl.so.1
shaftnode 13489 root txt VREG     32,0     68780  182342   /usr/lib/libsocket.so.1
shaftnode 13489 root txt VREG     32,0      2564  182324   /usr/lib/libdl.so.1
shaftnode 13489 root txt VREG     32,0    137160  182315   /usr/lib/ld.so.1
shaftnode 13489 root 0u   inet 0x507dc770 0t116   TCP   hostname:ftp->handler:53982
  (CLOSE_WAIT)
shaftnode 13489 root 1u   inet 0x507dc770 0t116   TCP   hostname:ftp->handler:53982
  (CLOSE_WAIT)
shaftnode 13489 root 2u   inet 0x507dc770 0t116   TCP   hostname:ftp->handler:53982
  (CLOSE_WAIT)
shaftnode 13489 root 3u   inet 0x5032c7d8    0t0   UDP   *:18753 (Idle)
```

Figure 2 - Information about a shaftnode agent, produced by lsof

As one can see, the agent is waiting to receive commands on its default UDP port number 18753. The TCP connection back to the handler remains unexplained to date.

Packet flows are listed in Figure 3.

There is considerable activity between the handler (`z.z.z.z`) and the agent (`x.x.x.x`) as they go through the command request and acknowledgement phases. A test of the impact of ICMP packet flooding on the local network itself may also occur, although the sheer volume of data preclude a listing in this paper.

Let us look at the individual phases from a later attack.

Setup and configuration phase

This is illustrated in Figure 4. The handler issues an "alive" command, and says "hi" to its agent, assigning a socket number of *5* and a ticket number of *8170*. We will see that this "socket number" will persist throughout this attack. A time period of 700 seconds is assigned to the agent, which is acknowledged. A packet size of 4096 bytes is specified, which is again confirmed. The last line indicates the type of attack, in this case "the works", i.e. UDP, TCP SYN and ICMP packet flooding combined. Failure to specify the type would make the agent default to UDP packet flooding.

Next is the list of hosts to attack and the ones from which statistics are to be obtained upon completion (Figure 5).

Now that all other parameters are set, the handler issues several *own* commands, in effect specifying the victim hosts. Those commands are acknowledged by the agent with an *owning* reply. The flooding occurs as soon as the first victim host gets added. The handler also requests packet statistics from the agents for certain victim hosts (e.g. *pktres tijgu 207.229.143.6 5 1993*). Note that the reply comes back with the same identifiers (*5 1993*) at the end of the 700 second packet flood, indicating that 51600 sets of packets were sent. One should realize that, if successful, this means 51600 x 3 packets due to the configuration of all three (UDP, TCP, and ICMP) types of packets. In turn, this results in roughly 220 4096 byte packets per second per host, or about 900 kilobytes per second per victim host from this agent alone, about 4.5 megabytes per second total for this little exercise.

Note the reverse shift ("shift" becomes "rghes", rather than "tijgu") for the password on the packet statistics.

Detection at the network level

Scanning the network for open port 20432 will reveal the presence of a handler on your LAN.

Date	Time	Protocol	Source IP/Port	Flow	Destination IP/Port
Sun 11/28	21:39:22	tcp	z.z.z.z.53982	<->	x.x.x.x.21
Sun 11/28	21:39:56	udp	x.x.x.x.33198	->	z.z.z.z.20433
Sun 11/28	21:45:20	udp	z.z.z.z.1765	->	x.x.x.x.18753
Sun 11/28	21:45:20	udp	x.x.x.x.33199	->	z.z.z.z.20433
Sun 11/28	21:45:59	udp	z.z.z.z.1866	->	x.x.x.x.18753
Sun 11/28	21:45:59	udp	x.x.x.x.33200	->	z.z.z.z.20433
Sun 11/28	21:45:59	udp	z.z.z.z.1968	->	x.x.x.x.18753
Sun 11/28	21:45:59	udp	z.z.z.z.1046	->	x.x.x.x.18753
Sun 11/28	21:45:59	udp	z.z.z.z.1147	->	x.x.x.x.18753
Sun 11/28	21:45:59	udp	z.z.z.z.1248	->	x.x.x.x.18753
Sun 11/28	21:45:59	udp	z.z.z.z.1451	->	x.x.x.x.18753
Sun 11/28	21:46:00	udp	x.x.x.x.33201	->	z.z.z.z.20433
Sun 11/28	21:46:00	udp	x.x.x.x.33202	->	z.z.z.z.20433
Sun 11/28	21:46:01	udp	x.x.x.x.33203	->	z.z.z.z.20433
Sun 11/28	21:48:37	udp	z.z.z.z.1037	->	x.x.x.x.18753
Sun 11/28	21:48:37	udp	z.z.z.z.1239	->	x.x.x.x.18753
Sun 11/28	21:48:37	udp	z.z.z.z.1340	->	x.x.x.x.18753
Sun 11/28	21:48:37	udp	z.z.z.z.1442	->	x.x.x.x.18753
Sun 11/28	21:48:38	udp	x.x.x.x.33204	->	z.z.z.z.20433
Sun 11/28	21:48:38	udp	x.x.x.x.33205	->	z.z.z.z.20433
Sun 11/28	21:48:38	udp	x.x.x.x.33206	->	z.z.z.z.20433
Sun 11/28	21:48:56	udp	z.z.z.z.1644	->	x.x.x.x.18753
Sun 11/28	21:48:56	udp	x.x.x.x.33207	->	z.z.z.z.20433
Sun 11/28	21:49:59	udp	x.x.x.x.33208	->	z.z.z.z.20433
Sun 11/28	21:50:00	udp	x.x.x.x.33209	->	z.z.z.z.20433
Sun 11/28	21:50:14	udp	z.z.z.z.1747	->	x.x.x.x.18753
Sun 11/28	21:50:14	udp	x.x.x.x.33210	->	z.z.z.z.20433

Figure 3 - Packet Flows

Date	Time	Source	Dest	Dest-port	Command
4 Dec 1999	18:06:40	z.z.z.z	x.x.x.x	18753	alive tijgu hi 5 8170
4 Dec 1999	18:09:14	z.z.z.z	x.x.x.x	18753	time tijgu 700 5 6437
4 Dec 1999	18:09:14	x.x.x.x	z.z.z.z	20433	time tijgu 5 6437 700
4 Dec 1999	18:09:16	z.z.z.z	x.x.x.x	18753	size tijgu 4096 5 8717
4 Dec 1999	18:09:16	x.x.x.x	z.z.z.z	20433	size tijgu 5 8717 4096
4 Dec 1999	18:09:23	z.z.z.z	x.x.x.x	18753	type tijgu 2 5 9003

Figure 4 - Set-up and Configuration

Date	Time	Source	Dest	Dest-port	Command
4 Dec 1999	18:09:24	z.z.z.z	x.x.x.x	18753	own tijgu 207.229.143.6 5 5256
4 Dec 1999	18:09:24	x.x.x.x	z.z.z.z	20433	owning tijgu 5 5256 207.229.143.6
4 Dec 1999	18:09:24	z.z.z.z	x.x.x.x	18753	pktres tijgu 207.229.143.6 5 1993
4 Dec 1999	18:09:24	z.z.z.z	x.x.x.x	18753	own tijgu 24.7.231.128 5 78
4 Dec 1999	18:09:24	z.z.z.z	x.x.x.x	18753	pktres tijgu 24.218.58.101 5 8845
4 Dec 1999	18:09:24	z.z.z.z	x.x.x.x	18753	own tijgu 18.85.13.107 5 6247
4 Dec 1999	18:09:25	z.z.z.z	x.x.x.x	18753	own tijgu 24.218.52.44 5 419
4 Dec 1999	18:09:25	z.z.z.z	x.x.x.x	18753	own tijgu 207.175.72.15 5 2376
4 Dec 1999	18:09:25	x.x.x.x	z.z.z.z	20433	owning tijgu 5 78 24.7.231.128
4 Dec 1999	18:09:26	x.x.x.x	z.z.z.z	20433	owning tijgu 5 6247 18.85.13.107
4 Dec 1999	18:09:27	x.x.x.x	z.z.z.z	20433	owning tijgu 5 4190 24.218.52.44
4 Dec 1999	18:09:28	x.x.x.x	z.z.z.z	20433	owning tijgu 5 2376 207.175.72.15
4 Dec 1999	18:21:04	x.x.x.x	z.z.z.z	20433	pktres rghes 5 1993 51600
4 Dec 1999	18:21:04	x.x.x.x	z.z.z.z	20433	pktres rghes 0 0 51400
4 Dec 1999	18:21:07	x.x.x.x	z.z.z.z	20433	pktres rghes 0 0 51500
4 Dec 1999	18:21:07	x.x.x.x	z.z.z.z	20433	pktres rghes 0 0 51400
4 Dec 1999	18:21:07	x.x.x.x	z.z.z.z	20433	pktres rghes 0 0 51400

Figure 5 - Host Attack Specifications

For detecting idle agents, one could write a program similar to George Weaver's trinoo detector. Sending out "alive" messages with the default password to all nodes on a network on the default UDP port 18753 will generate traffic back to the detector, making the agent believe the detector is a handler.

This program does not provide for code updates (like TFN or Stacheldraht). This may imply "rcp" or "ftp" connections during the initial intrusion phase (see also [5]).

The program uses UDP traffic for its communication between the handlers and the agents. Considering that the traffic is not encrypted, it can easily be detected based on certain keywords. Performing an "ngrep" [11] for the keywords mentioned in the syntax sections above, will locate the control traffic, and looking for TCP packets with sequence numbers of 0x28374839 may locate the TCP SYN packet flood traffic. Source ports are always above 1024, and source IP numbers can include zeroes in the leading octet.

Strings in this control traffic can be detected with the "ngrep" program using the same technique shown in [5], [6], and [7]. For example,

```
# ngrep -i -x "alive tijgu" udp
# ngrep -i -x "pktres|pktstat" udp
```

will locate the control traffic between the handler and the agent, independently of the port number used.

There are also two excellent scanners for detecting DDoS agents on the network: Dittrich's *dds* [8] and Brumley's *rid* [2].

dds was written to provide a more portable and less dependent means of scanning for various DDoS tools. (Many people encountered problems with Perl and the Net::RawIP library [15] on their systems, which prevented them from using the scripts provided in [5], [6], and [7].) Due to time contraints during coding, dds does not have the flexibility necessary to specify arbitrary protocols, ports, and payloads. A modified version of dds, geared towards detecting only Shaft agents, can be found at:

http://sled.gsfc.nasa.gov/~spock/

A better means of detecting Shaft handlers and agents would be to use a program like rid, which uses a more flexible configuration file mechanism to define ports, protocols, and payloads.

A sample configuration for rid to detect the Shaft control traffic as described:

```
start shaft
  send udp dport=18753 data="alive
  tijgu hi 5 1918"
  recv udp sport=20433 data="alive"
  nmatch=1
end shaft
```

Defenses

To protect against the effects of the multiple types of denial of service, we suggest that you review the other papers (see [1, 3, 5, 6, 7]) and other methods of dealing with DDoS attacks being discussed and promoted (see [9]). For example, rate-limiting is considered effective against ICMP packet flooding attacks, while anti-spoof filters and egress filters at the border routers can limit the problems caused by attacking agents faking source addresses. Regular scanning for the presence of DDoS tools is another excellent strategy.

Further evolution

While the author(s) of this tool did not pursue the use of encryption of its control traffic, such an evolution is conceivable, since a Caesar cipher is used to obfuscate the password. A transition to Blowfish or other stream ciphers is realistic, and changing the communication protocol to ICMP, much like TFN, is conceivable. The use of multicast protocols for both communication and packet flooding is also possible.

To date, no source code for the Shaft handler (shaftmaster) has been obtained or analyzed.

At this stage, we believe that the code is private. This would mean that the authors could likely change defaults; the probability of detecting "script kiddie" copycats using default values as analyzed here is low. This provides strong impetus for rapid and widespread detection efforts to identify agents before this change.

Conclusion

Shaft is another DDoS variant with independent origins. The code recovered appeared to be still in development. Several key features indicate evolutionary trends as the genre develops. Of significance is the priority placed on packet generation statistics which would allow host selection to be refined. The analysis of the code and binary was greatly enhanced by the capture of attack preparation and command packets. The captured packets made it possible to assess the impact of a single agent that managed to saturate the network pipe. The version analyzed had hooks which would allow for dynamic changes to the master host and control port but not the agent control port. However such items are trivially incorporated and must not be taken to be indicative of any current versions which may be in active use. The obfuscation of master IP, ports and passwords used a relatively simple form of encryption but this could easily be strengthened.

The detection of DDoS installations will become very much more difficult as such metamorphosis techniques progress, the presence of such agents will still be more readily determined by analysis of traffic anomalies with a consequent pressure on time and resources for site administrators and security teams.

References

[1] Barlow, Jason and Woody Thrower. TFN2K - An Analysis
http://www2.axent.com/swat/News/TFN2k_Analysis.htm

[2] Brumley, David. Remote Intrusion Detector.
http://theorygroup.com/Software/RID

[3] CERT Distributed System Intruder Tools Workshop Report
http://www.cert.org/reports/dsit_workshop.pdf

[4] CERT Advisory CA-99-17 Denial-of-Service Tools
http://www.cert.org/advisories/CA-99-17-denial-of-service-tools.html

[5] Dittrich, David. The DoS Project's trinoo distributed denial of service attack tool
http://staff.washington.edu/dittrich/misc/trinoo.analysis

[6] Dittrich, David. The "Tribe Flood Network" distributed denial of service attack tool
http://staff.washington.edu/dittrich/misc/tfn.analysis

[7] Dittrich, David. The "Stacheldraht" distributed denial of service attack tool
http://staff.washington.edu/dittrich/misc/stacheldraht.analysis

[8] Dittrich, David, Marcus Ranum, George Weaver, David Brumley et al.
http://staff.washington.edu/dittrich/dds

[9] Dittrich, David, Distributed Denial of Service (DDoS) Attacks/Tools
http://staff.washington.edu/dittrich/misc/ddos/

[10] lsof:
http://vic.cc.purdue.edu/

[11] ngrep:
http://www.packetfactory.net/Projects/ngrep/

[12] Packet Storm Security, Distributed denial of service attack tools
http://packetstorm.securify.com/distributed/

[13] Phrack Magazine, Volume Seven, Issue Forty-Nine, File 06 of 16, [Project Loki]
http://www.phrack.com/search.phtml?view&article=p49-6

[14] Phrack Magazine Volume 7, Issue 51 September 01, 1997, article 06 of 17 [LOKI2 (the implementation)]
http://www.phrack.com/search.phtml?view&article=p51-6

[15] Net::RawIP:
http://quake.skif.net/RawIP

[16] tcpdump:
ftp://ftp.ee.lbl.gov/tcpdump.tar.Z

[17] Schneier, Bruce. Applied Cryptography, 2nd edition, Wiley.

[18] Stevens, W. Richard and Gary R. Wright. TCP/IP Illustrated, Vol. I, II, and III., Addison-Wesley.

[19] Zuckerman, M.J. Net hackers develop destructive new tools. USA Today, 7 December 1999.
http://www.usatoday.com/life/cyber/tech/review/crg681.htm

Dr. Sven Dietrich is a Senior Security Architect working for Raytheon ITSS at the NASA Goddard Space Flight Center. He received a B.Sc. in Mathematics and Computer Science, a M.S. and a Doctor of Arts in Mathematics from Adelphi University, New York. Dr. Dietrich's primary efforts focus on deployment of public-key technology, the building of a public-key infrastructure (PKI) for NASA, intrusion detection, and the security of Internet Protocol (IP) communications in space. For his involvement in the latter he has recently received the NASA GSFC National Resource Group Achievement Award. Before joining Raytheon ITSS in 1997, he served on the faculty at Adelphi University for six years. He is actively involved in the computer security field inside and outside of NASA and randomly gives presentations and talks on the subject.

Dr. Neil Long is a Senior Systems Administrator at Oxford University, England and a former academic (Ph.D., B.Sc. Materials Science) who has been involved in computer security incident handling for about 8 years. He is a member of OxCERT (Oxford University computer emergency response team and member of FIRST) and serves on the steering committee member for the FIRST organisation (Forum of Incident Response and Security Teams). Since the career switch there has been little need to publish although he has given several presentations and seminars and views computer security as an endless source of new research material.

Dave Dittrich is a Senior Security Consultant at the University of Washington, supporting Unix workstation users on campus. Dave has spoken at various user groups and computer conferences from Seattle, Washington to as far away as Darwin, Australia. In his spare time, Dave enjoys photography (a side business) and telemark skiing (ski mountaineering and racing.) Dave is most widely known for his technical analyses of the Trinoo, Tribe Flood Network, and Stacheldraht distributed denial of service attack tools. His home page can be found at http//staff.washington.edu/dittrich